

AFRL-IF-RS-TR-2003-164
Final Technical Report
July 2003



SPECIFYING AND CHECKING SECURITY PROPERTIES IN AN EVOLVING SOFTWARE BASE

Massachusetts Institute of Technology

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. G377

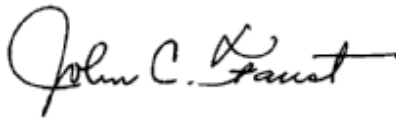
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.


The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-164 has been reviewed and is approved for publication.

APPROVED: 
JOHN C. FAUST
Project Engineer

FOR THE DIRECTOR: 
WARREN H. DEBANY JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE JULY 2003		3. REPORT TYPE AND DATES COVERED Final May 98 – Sep 02
4. TITLE AND SUBTITLE SPECIFYING AND CHECKING SECURITY PROPERTIES IN AN EVOLVING SOFTWARE BASE			5. FUNDING NUMBERS C - F30602-98-1-0237 PE - 62301E PR - G377 TA - 10 WU - 01	
6. AUTHOR(S) Barbara H. Liskov				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Massachusetts Institute of Technology MIT Laboratory for Computer Science 545 Technology Square Cambridge Massachusetts 02139-4307			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/IFGB 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2003-164	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: John C. Faust/IFGB/(315) 330-4544/ John.Faust@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) Research funded under this grant led to the development of techniques and tools for protecting privacy in a decentralized environment and achieving Byzantine Fault Tolerance (BFT), and a methodology that enables BFT replicas to run different implementations. The work led to an innovative new security model that allows static checking of security properties, a new annotation language for expressing security properties, extensions to Java that allow code to use the new model, lightweight tools for checking security properties of both source code (via a new compiler) and byte codes (via a new bytecode verifier), and a study of runtime support needed by the model. A new replications algorithm (BFT) that is robust against Byzantine failures was developed. It is efficient, works in an asynchronous environment and can be used to harden critical system services. An extension to BFT, called BFT with Abstract Specification Encapsulation (BASE), was developed to allow different software to run at different replicas so as to avoid failures due to software bugs. It provides a way of achieving practical N-version programming in which different versions are developed by different organizations and also the different versions may differ in the details of their behavior, i.e., support slightly different specifications.				
14. SUBJECT TERMS Information Flow, Decentralized Label Model, Declassification, Java Information Flow, Byzantine Fault Tolerance, Abstract Specification Encapsulation, Conformance Wrapper				15. NUMBER OF PAGES 22
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

1. Introduction.....	1
2. Results.....	1
2.1 Protecting Privacy in a Decentralized Environment.....	1
2.1.1 Privacy Example	2
2.1.2 Label Model.....	3
2.1.3 Privacy Example Revisited.....	6
2.2 Supporting the Security Model.....	7
2.2.1 The JIF Language	7
2.2.2 Trusted Execution Platform	8
2.3 Byzantine Fault Tolerance	9
2.3.1 The BFT Algorithm	9
2.3.2 Implementation	12
2.4 BASE: Using Abstraction to Improve Fault Tolerance	12
2.4.1 Implementation	14
3. Technology Transfer.....	14
3.1 Software	14
3.2 Personnel.....	14
4. Publications.....	15
4.1 Information Flow Publications	15
4.2 Byzantine-Fault-Tolerance Publications.....	15
5. References.....	17

List of Figures

Figure 1: A Simple Example	2
Figure 2: A Principle Hierarchy.....	4
Figure 3: Annotated Tax Preparation Example.....	6
Figure 4: Implicit Information Flow.....	7
Figure 5: Normal Case Operation.....	11

1 Introduction

The research funded under this grant led to the development of techniques and tools that, for the first time ever, allow practical control of privacy of information. The work led to an innovative new security model that allows static checking of security properties, a new annotation language for expressing security properties statically, extensions to Java that allow code to use the new model, lightweight tools for checking security properties of both source code (via a new compiler) and bytecodes (via a new bytecode verifier), and a study of runtime support needed by the model, in particular what is needed to provide a trusted execution platform that runs imported code on imported data while ensuring the privacy of both local and imported information.

Our security model depends on a principal hierarchy that contains information about relationships among principals, e.g., whether principal A can act for principal B. It is vital that the principal hierarchy not be corrupted, even by a malicious attack. No existing algorithms provide this ability in an efficient and general way.

Our work also addresses this problem. We have developed BFT, a new replication algorithm that is robust against Byzantine failures, yet is efficient and works in an asynchronous environment. The new algorithm can be used to harden the principal hierarchy against attack; it can also be used to harden other important services, such as DNS.

We also developed an extension to BFT, called BASE. BASE takes advantage of work in programming methodology and data abstraction to allow systems to make use of BFT even though the code running at the replicas is not identical. It allows different software to run at different replicas, e.g., different implementations of NFS, so as to avoid failures due to software bugs. This approach provides a way of achieving practical N-version programming in which the different versions are developed by different organizations (so that they are truly independent) and also the different versions may differ in the details of their behavior, i.e., support slightly different specifications.

2 Results

2.1 Protecting Privacy in a Decentralized Environment

The common models for computer security are proving inadequate. Security models have two goals: preventing accidental or malicious destruction of information, and controlling the release and propagation of that information. Only the first of these goals is supported well at present, by security models based on access control lists or capabilities. Access control mechanisms do not support the second goal well: they help to prevent information release but do not control information propagation. For example, if user *A* is allowed to read *B*'s data, *B* cannot control how *A* distributes the information it has read. Control of information propagation *is* supported by existing information flow and compartmental models [2, 7], but these models unduly restrict the computation that can be performed.

Information flow control is vital whenever there is downloading of distrusted code, as would happen at a router in an active network. Java [10] could be used to download the code and it attempts to prevent the downloaded code from transferring private data to other sites by using a compartmental *sandbox* security model. However, this approach largely prevents applications from sharing data and it does not support fine-grained sharing in which different data manipulated by an application have different security requirements.

Our research led to the development of an approach that provides the needed support. The basis of the work is the *decentralized label model*, a new model of decentralized information flow

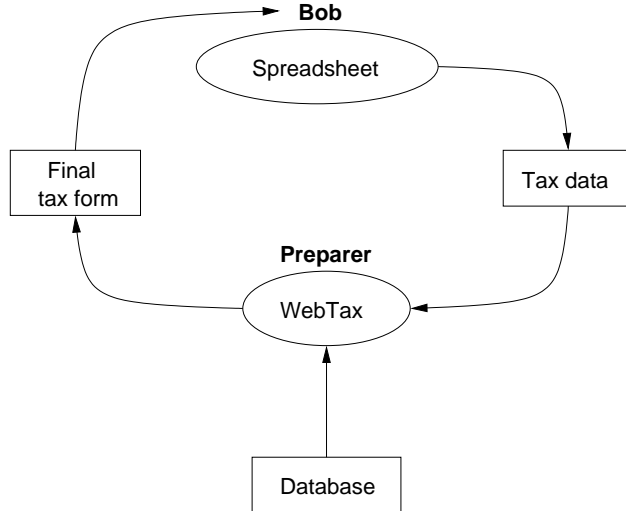


Figure 1: A simple example

that relaxes the restrictions of previous models yet is inexpensive in both space and time. The new model supports fine-grained information sharing between distrusted applications, while reducing the potential for information leaks. It allows users to control the flow of their information without imposing the rigid constraints of a traditional multilevel security system. It provides security guarantees to users and to groups rather than to a monolithic organization. It differs from previous work on information flow control by allowing users to explicitly *declassify* (or *downgrade*) data that they own. When data is derived from several sources, all the sources must agree to release the data.

This section provides an overview of our label model; more details can be found in [21, 25, 24, 22, 26, 23].

2.1.1 Privacy Example

Figure 1 depicts an example with security requirements that cannot be satisfied using existing techniques. This scenario contains mutually distrusting principals that must cooperate to perform useful work. In the example, the user Bob is preparing his tax form using both a spreadsheet program and a piece of software called “WebTax”. Bob would like to be able to prepare his final tax form using WebTax, but he does not trust WebTax to protect his privacy. The computation is being performed using two programs: a spreadsheet that Bob trusts and to which he grants his full authority, and the WebTax program, which Bob does not trust. Bob would like to transmit his tax data from the spreadsheet to WebTax and receive a final tax form as a result, while being protected against WebTax leaking his tax information.

In this example, there is another principal named Preparer that also has privacy interests. This principal represents a firm that distributes the WebTax software. The WebTax application computes the final tax form using a proprietary database, shown at the bottom, that is owned by Preparer. This database might, for example, contain secret algorithms for minimizing tax payments. Since this principal is the source of the WebTax software, it trusts the program not to distribute the proprietary database through malicious action, though the program might leak information because it contains bugs.

It may be difficult to prevent some information about the database contents from leaking back to Bob, particularly if Bob is able to make a large number of requests and then carefully analyze the resulting tax forms. This information leak is not a practical problem if Preparer can charge Bob a per-form fee that exceeds the value of the information Bob obtains through each form.

To make this scenario work, Preparer needs two pieces of functionality. First, it needs protection against accidental or malicious release of information from the database by paths other than through the final tax form. Second, it needs the ability to *sign off* on the final tax form, confirming that the information leaked in the final tax form is sufficiently small or scrambled by computation that the tax form may be released to Bob.

It is worth noting that Bob and Preparer do need to trust that the execution platform has not been subverted. For example, if WebTax is running on a computer that Bob controls completely, Bob will be able to steal the proprietary database. Clearly, Preparer cannot have any real expectation of privacy or secrecy if its private data is manipulated in unencrypted form by an execution platform that it does not trust!

However, even if the execution platform is trusted, the scenario cannot be implemented satisfactorily using existing security techniques. With current techniques, Bob must inspect the WebTax code and verify that it does not leak his data; in general, this task is impossible. Our techniques allow the security goals of both Bob and Preparer to be met without this inspection; Bob and Preparer can then cooperate in performing useful computation. In another sense, this work shows how both Bob and Preparer can inspect the WebTax program efficiently and simply to determine whether it violates their security requirements.

2.1.2 Label Model.

We provide security through the *decentralized label model*. The key new feature of this model is that it supports computation in an environment with mutual distrust. The ability to handle mutual distrust is achieved by attaching a notion of ownership to information flow policies. These policies can then be modified by their owners—a form of safe declassification. Declassification is safe because flow policies of other principals remain in force.

The essentials of the decentralized label model are *principals*, which are the entities whose privacy is protected by the model; and *labels*, which are the way that principals express their privacy concerns. In addition, there are rules that must be followed as computation proceeds in order to avoid information leaks, including the safe declassification mechanism. These rules have been shown to be both sound and complete with respect to a simple formal semantics for labels: the rule only allows safe relabellings, and it allows *all* safe relabellings [24].

In this model, users are assumed to be external to the system on which programs run. Information is leaked only when it leaves the system. The model also contains rules governing the release of information to the external environment, as well as rules for reading information from the environment.

Principals. Information is owned by, updated by, and released to *principals*: users and other authority entities such as groups or roles. For example, both users and groups in Unix would be modeled as principals. A process has the authority to act on behalf of some set of principals.

Some principals are authorized to *act for* other principals. When one principal p can act for another principal p' , p possesses all the potential powers and privileges of p' . The *acts for* relation is reflexive and transitive, defining a hierarchy or partial order of principals. For compactness, the statement “ p acts for q ” is written formally as $p \succeq q$. This relation is similar to the *speaks for*

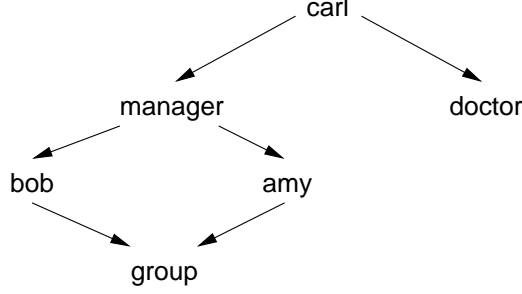


Figure 2: A principal hierarchy

relation [17]; the principal hierarchy also is similar to a *role hierarchy* [30]. Although the principal hierarchy changes over time, revocation of acts-for relations is assumed to occur infrequently.

The acts-for relation can be used to model groups and roles conveniently, as shown in Figure 2. In this example, a group named “group” is modeled by introducing acts-for relations between each of the group members (“amy” and “bob”) and the group principal. This relation allows Amy and Bob to read data readable by the group and to control data controlled by the group. A principal “manager” representing Bob and Amy’s manager is able to act for both “amy” and “bob”. This principal is one of the roles that a third user, Carl (“carl”), is currently enabled to fulfill; Carl also has a separate role as a doctor. Carl can use his roles to prevent accidental leakage of information between data stores associated with his different jobs.

The notion of the acts-for relation permits delegation of all of a principal’s powers, or none, but it can be extended to support finer-grained principal relations in a natural manner [22].

Labels. Principals express their privacy concerns by using labels to annotate programs and data. A label is a set of *policies* that express privacy requirements. A policy has two parts, an owner and a set of readers, and is written in the form *owner: readers*. The owner of a policy is a principal whose data was observed in order to construct the value labeled by this policy. The readers of a policy are the principals that this policy permits to read the data. The owner is a source of data, and the readers are possible destinations for the data. Principals not listed as readers are not permitted to read the data.

An example of a label is $L = \{o_1 : r_1, r_2; o_2 : r_2, r_3\}$. Here, o_1, o_2, r_1, r_2, r_3 denote principals. Semicolons separate two policies within the label L . The owners of these policies are o_1 and o_2 , and the reader sets of the policies are $\{r_1, r_2\}$ and $\{r_2, r_3\}$, respectively.

The intuitive meaning of a label is that every policy in the label must be obeyed as data flows through the system, so that labeled information is released only by the consensus of all of the policies. A user may read the data only if the user’s principal can act for a reader for *each* policy in the label. Thus only users whose principals can act for r_2 can read the data labeled by L .

The same principal may be the owner of several policies; the policies are still enforced independently of one another in this case. If a principal is the owner of *no* policies in the label, it is as if the label contained a policy with that owner, but every possible principal listed as a reader.

This label structure allows each owner to specify an independent flow policy, and thus to retain control over the dissemination of its data. Code running with the authority of an owner can modify the flow policy for a policy with that owner; in particular, the program can *declassify* that data by adding additional readers. Since declassification applies on a per-owner basis, no centralized declassification process is needed, as it is in systems that lack ownership labeling.

Labels are used to control information flow within programs by annotating program variables with labels. The label of a variable controls how the data stored within that variable can be disseminated. The key to secure flow is to ensure that as data flows through the system during computation, its labels only become more restrictive: the labels have more owners, or particular owners allow fewer readers. If the contents of one variable affect the contents of another variable, e.g., the second variable receives the results of a computation based on the first variable, there is an information flow from the first variable to the second, and therefore, the label of the second variable must be at least as restrictive as the label of the first.

This condition can be enforced at compile time as a kind of type checking, as long as the label of a variable is known at compile time. In other words, the label of a variable cannot change at run time, and the label of the data contained within a variable is always the same as the label of the variable itself. If the label of a variable could change, the label would need to be stored and checked at run time, creating an additional information channel and leading to significant performance overhead. Immutable variable labels might seem like a limitation, but full expressiveness can be obtained by having variables that contain labels.

When a value is read from a variable x , the apparent label of the value is the label of x ; whatever label that value had at the time it was written to x is no longer known when it is read. In other words, writing a value to a variable is a *relabeling*, and is allowed only when the label of the variable is at least as restrictive as the apparent label of the value being written.

Giving private data to an untrusted program does not create an information leak—even if that program runs with the authority of another principal—as long as that program obeys all of the label rules described here. Information can be leaked only when it leaves the system through an *output channel*, so output channels are labeled to prevent leaks. Information can enter the system through an *input channel*, which also is labeled to prevent leaks. It is safe for a process to manipulate data even though no principal in its authority has the right to read it, because all the process can do is write the data to a variable or a channel with a label that is at least as restrictive as the data’s label.

Declassification. A very important part of the model is its support for *declassification*: owners can relax restrictions on data they own when appropriate.

The ability of a process to declassify data depends on the authority possessed by the process. At any moment while executing a program, a process is authorized to act on behalf of some (possibly empty) set of principals. This set of principals is referred to as the *authority* of the process. If a process has the authority to act for a principal, actions performed by the process are assumed to be authorized by that principal. Code running with the authority of a principal can declassify data by creating a copy in whose label a policy owned by that principal is relaxed. In the label of the copy, readers may be added to the reader set, or the policy may be removed entirely, which is effectively the same as adding all principals as readers in the policy.

Because declassification applies on a per-owner basis, no centralized declassification process is needed, as it is in systems that lack ownership labeling. Declassification is limited because it cannot affect the policies of owners the process does not act for and therefore it is safe for the other owners.

The declassification mechanism makes it clear why the labels maintain independent reader sets for each owning principal. If, instead, a label consisted of just an owner set and a reader set, information about the individual flow policies would be lost, reducing the power of declassification.

Because the ability to declassify depends on the run-time authority of the process, it requires a run-time check for the proper authority. Declassification seems to be needed infrequently, so that the overhead of this run-time check is not large. It can be reduced still further using the proper

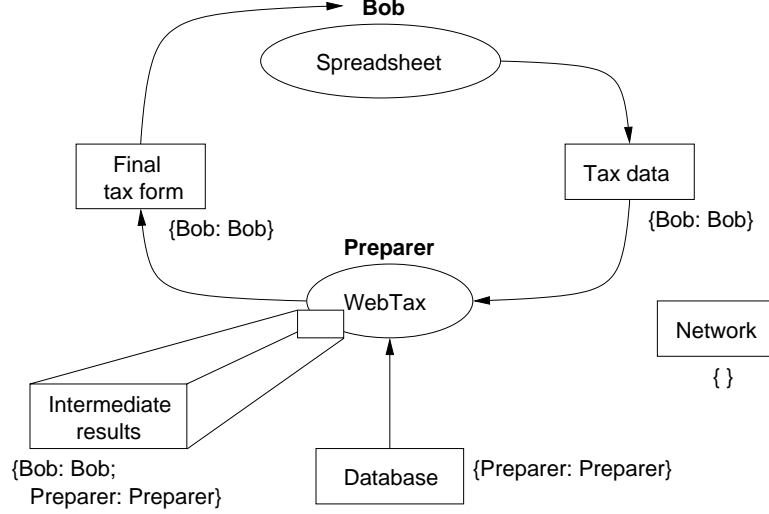


Figure 3: Annotated Tax Preparation Example

static framework [21].

2.1.3 Privacy Example Revisited

The tax preparer example from Section 2.1.1 is repeated in Figure 3, except that all data in the example has been annotated with labels to protect the privacy of the principals Bob and Preparer. It can be seen that these labels obey the rules given and meet the security goals set out for this scenario.

In the figure, ovals indicate programs executing in the system. A boldface label beside an oval indicates the authority with which a program acts. In this example, the principals involved are “Bob” and “Preparer”, as we have already seen, and they give their authority to the spreadsheet and WebTax programs, respectively. Arrows in the diagrams represent information flows between principals; square boxes represent information that is flowing, or databases of some sort.

First, Bob applies the label $\{\text{"Bob": "Bob"}\}$ to his tax data. This label allows no one to read the data except Bob himself. With this label applied to it, tax data cannot be sent to an untrusted network location, represented as an output channel with label $\{\}$. Bob can give this data to the WebTax program with confidence that it cannot be leaked, because WebTax will be unable to remove the $\{\text{"Bob": "Bob"}\}$ policy from the tax data or any data derived from it.

The WebTax program uses Bob’s tax data and its private database to compute the tax form. Any intermediate results computed from these data sources will have the label $\{\text{"Bob: Bob"; "Preparer: Preparer"}\}$, since computations retain all labels of their inputs. Because the reader sets of this label disagree, the label prevents both Bob and Preparer (and everyone else) from reading the intermediate results. Preparer is protected by this label against accidental disclosure of its private database through programming errors in the WebTax application.

Before being released to Bob, the final tax form has the same label as the intermediate results, and is not readable by Bob, appropriately. To make the tax form readable, the WebTax application *declassifies* the label by removing the $\{\text{"Preparer": "Preparer"}\}$ policy. The application can do this because the “Preparer” principal has granted the application its authority. This grant of authority is reasonable because “Preparer” supplied the application and presumably trusts that it will not

```

x = 0;
if (b) {
    x = 1;
}

```

Figure 4: Implicit information flow

use the power maliciously.

The authority to act as “Preparer” need not be possessed by the entire WebTax application, but only by the part that performs the final release of the tax form. By limiting this authority to a small portion of the application, the risk of accidental release of the database is reduced. Thus the WebTax application might have a small top-level routine that runs with the authority of “Preparer” while the rest of its code runs with no authority.

2.2 Supporting the Security Model

This section briefly describes the other parts of our research on information flow: the JIF language, and the *trusted execution platform*, a runtime environment that ensures that application code is run in a way that avoids information leaks.

2.2.1 The JIF Language

Incorporating the model in a programming language makes it easy for programmers to write code that is label correct and also enables compile time checking of information flow. The model can be incorporated into any programming language; we chose to extend Java because it is becoming widely used. The resulting language is called JIF, which stands for “Java Information Flow.”

The idea behind JIF is that annotated programs are checked statically to verify that all information flows within the program follow the rules given in the previous section. Information flow checks can be viewed as an extension to type checking. For both kinds of static analysis, the compiler determines that certain operations are not permitted to be performed on certain data values. The analogy goes further, because in a language with subtyping one can distinguish between notions of the apparent (static) and actual (run-time) type of a value. Similarly, static label checking can be considered as a conservative approximation to checking the actual run-time labels.

However, there is a difficulty in checking labels at run time that does not arise with type checking. Type checks may be performed at compile time or at run time, though compile-time checks are obviously preferred when applicable because they impose no run-time overhead. By contrast, fine-grained information flow control is practical only with static analysis, which may seem odd; after all, any check that can be performed by the compiler can be performed at run time as well. The difficulty with run-time checks is the fact that they can *fail*. In failing, they may communicate information about the data that the program is running on. Unless the information flow model is properly constructed, the fact of failure (or its absence) can serve as a covert channel. By contrast, the failure of a compile-time check reveals no information about the actual data passing through a program. A compile-time check only provides information about the program that is being compiled. Similarly, link-time and load-time checks provide information only about the program, and may be considered to be static checks for the purposes of this work.

Implicit information flow [8] is difficult to prevent without static analysis. For example, consider the segment of code shown in Figure 4 and assume that the storage locations *b* and *x* belong to different security classes B and X, respectively, and B is more sensitive than X, so data should not

flow from b to x . However, the code segment stores 1 into x if b is true, and 0 into x if b is false; x effectively contains the value of b . A run-time check can detect easily that the assignment $x = 1$ communicates information improperly, and abort the program at this point. Consider, however, the case where b is false: no assignment to x occurs within the context in which b affects the flow of control. The fact of the program's aborting or continuing implicitly communicates information about the value of b , which can be used in at least the case where b is false.

Static analysis would be too limiting for structures like file systems, where information flow cannot be verified statically. JIF defines a new secure run-time escape hatch for these structures, with explicit run-time label checks. Uses of the run-time information flow mechanism are still verified statically, to ensure that they do not leak information.

JIF extends previous work on static flow checking in two ways. First, it contains powerful language features that have not been integrated with static flow checking previously, including mutable objects, subclassing, dynamic type tests, and exceptions. Second, it provides powerful new features that make information flow checking less restrictive and more convenient than in previous programming languages:

- A simple but powerful model of authority allows code privileges to be checked statically, and also allows authority to be granted and checked dynamically.
- *Label polymorphism* permits code that is generic with respect to the security class of the data it manipulates.
- Run-time label checking and first-class label values provide an escape to run-time checks when static checking is too restrictive. Run-time checks are statically checked to ensure that information is not leaked by the success or failure of the run-time check itself.
- Automatic label inference makes unnecessary many of the annotations that otherwise would be required.

2.2.2 Trusted Execution Platform

To ensure that programs preserve privacy, it is not enough to check their label-correctness at compile time; we also need to ensure that their execution is carried out in a way that is consistent with the labels. Therefore we need to provide a runtime environment that runs programs in a label correct way. We call this environment the *trusted execution platform* [1]. This platform might run on a router in an active network, or more generally on some node in the Internet.

The platform allows code to be imported and ensures that the imported code follows the rules. For example, it can import the tax preparer's code and also code provided by Bob and ensure that neither program exposes the other's private information.

The platform runs two kinds of code, trusted code and untrusted code. All code that it runs for users, e.g., Bob's code and the tax preparer's code, is untrusted.

The trusted computing base, which includes all the trusted code, must include a mechanism that ensures that untrusted code follows the rules of the model. The mechanism might be a compiler that statically checks the information flows in a program and then digitally signs the program. However, having the compiler in the trusted base is not desirable since compilers are very large. Therefore, the platform runs a verifier that checks the output of such a compiler (as annotated byte codes).

The trusted base must also include many of the usual trusted components of an operating system: hardware that has not been subverted, a trustworthy underlying operating system, and a reliable authentication mechanism.

2.3 Byzantine Fault Tolerance

Our research under this grant led to the development of a new replication protocol that offers both integrity and high availability in the presence of Byzantine faults. Our replication technique allows a system to continue to function correctly even when some replicas are compromised by an attacker; this is worthwhile because the growing reliance on online information services makes malicious attacks more likely and their consequences more serious. The system also survives nondeterministic software bugs and software bugs due to aging (e.g., memory leaks). Our approach improves on the usual technique of rebooting the system because it refreshes state automatically, staggers recovery so that individual replicas are highly unlikely to fail simultaneously, and has little impact on overall system performance.

Because of recovery, our system can tolerate any number of faults over the lifetime of the system, provided fewer than $1/3$ of the replicas become faulty within a window of vulnerability. The best that could be guaranteed previously was correct behavior if fewer than $1/3$ of the replicas failed during the lifetime of a system. Our previous work [4] guaranteed this and other systems [29, 14] provided weaker guarantees. Limiting the number of failures that can occur in a finite window is a synchrony assumption but such an assumption is unavoidable: since Byzantine-faulty replicas can discard the service state, we must bound the number of failures that can occur before recovery completes. But we require no synchrony assumptions to match the guarantee provided by previous systems.

The window of vulnerability can be small (e.g., a few minutes) under normal conditions. Additionally, our algorithm provides *detection* of denial-of-service attacks aimed at increasing the window: replicas can time how long a recovery takes and alert their administrator if it exceeds some pre-established bound. Therefore, integrity can be preserved even when there is a denial-of-service attack.

The remainder of this section provides an overview of our approach. More details can be found in [5].

2.3.1 The BFT Algorithm

Our algorithm is a form of *state machine* replication [15, 31]: the service is modeled as a state machine that is replicated across different nodes in a distributed system. The algorithm can be used to implement any replicated *service* with a *state* and some *operations*. The operations are not restricted to simple reads and writes; they can perform arbitrary computations.

The service is implemented by a set of replicas \mathcal{R} and each replica is identified using an integer in $\{0, \dots, |\mathcal{R}| - 1\}$. Each replica maintains a copy of the service state and implements the service operations. For simplicity, we assume $|\mathcal{R}| = 3f + 1$ where f is the maximum number of replicas that may be faulty. Service clients and replicas are non-faulty if they follow the algorithm and if no attacker can impersonate them (e.g., by forging their MACs).

Like all state machine replication techniques, we impose two requirements on replicas: they must start in the same state, and they must be *deterministic* (i.e., the execution of an operation in a given state and with a given set of arguments must always produce the same result).

Our algorithm ensures safety for an execution provided at most f replicas become faulty within a window of vulnerability of size T_v . Safety means that the replicated service satisfies linearizability [12, 3]: it behaves like a centralized implementation that executes operations atomically one at a time. Our algorithm provides safety regardless of how many faulty clients are using the service (even if they collude with faulty replicas).

The algorithm also guarantees liveness: non-faulty clients eventually receive replies to their requests provided (1) at most f replicas become faulty within the window of vulnerability T_v ; and (2) denial-of-service attacks do not last forever, i.e., there is some unknown point in the execution after which all messages are delivered (possibly after being retransmitted) within some constant time d , or all non-faulty clients have received replies to their requests. Here, d is a constant that depends on the timeout values used by the algorithm to refresh keys, and trigger view-changes and recoveries.

The algorithm works as follows. Clients send requests to execute operations to the replicas and all non-faulty replicas execute the same operations in the same order. Since replicas are deterministic and start in the same state, all non-faulty replicas send replies with identical results for each operation. The client waits for $f + 1$ replies from different replicas with the same result. Since at least one of these replicas is not faulty, this is the correct result of the operation.

The hard problem is guaranteeing that *all non-faulty replicas agree on a total order for the execution of requests despite failures*. We use a primary-backup mechanism to achieve this. In such a mechanism, replicas move through a succession of configurations called *views*. In a view one replica is the *primary* and the others are *backups*. We choose the primary of a view to be replica p such that $p = v \bmod |\mathcal{R}|$, where v is the view number and views are numbered consecutively.

The primary picks the ordering for execution of operations requested by clients. It does this by assigning a sequence number to each request. But the primary may be faulty. Therefore, the backups trigger *view changes* when it appears that the primary has failed to select a new primary. Viewstamped Replication [28] and Paxos [16] use a similar approach to tolerate benign faults.

To tolerate Byzantine faults, every step taken by a node in our system is based on obtaining a *certificate*. A certificate is a set of messages certifying some *statement* is correct and coming from different replicas. An example of a statement is: “the result of the operation requested by a client is r ”.

The size of the set of messages in a certificate is either $f + 1$ or $2f + 1$, depending on the type of statement and step being taken. The correctness of our system depends on a certificate never containing more than f messages sent by faulty replicas. A certificate of size $f + 1$ is sufficient to prove that the statement is correct because it contains at least one message from a non-faulty replica. A certificate of size $2f + 1$ ensures that it will also be possible to convince other replicas of the validity of the statement even when f replicas are faulty.

We use a three-phase protocol to atomically multicast requests to the replicas. The three phases are *pre-prepare*, *prepare*, and *commit*. The pre-prepare and prepare phases are used to totally order requests sent in the same view even when the primary, which proposes the ordering of requests, is faulty. The prepare and commit phases are used to ensure that requests that commit are totally ordered across views. Figure 5 shows the operation of the algorithm in the normal case of no primary faults.

View Changes. The view-change protocol provides liveness by allowing the system to make progress when the primary fails. View changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute. A backup is waiting for a request if it received a valid request and has not executed it. A backup starts a timer when it receives a request and the timer is not already running. It stops the timer when it is no longer waiting to execute the request, but restarts it if at that point it is waiting to execute some other request.

If the timer of backup i expires in view v , the backup starts a view change to move the system to view $v + 1$. It stops accepting messages for the normal execution of client requests and multicasts a VIEW-CHANGE message to all replicas.

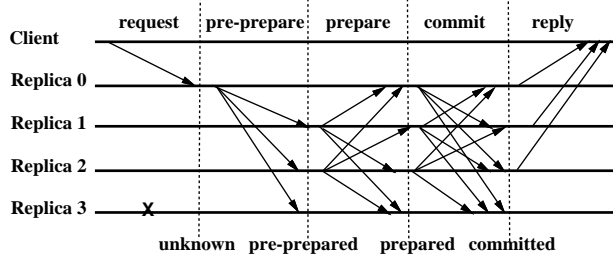


Figure 5: Normal Case Operation. Replica 0 is the primary, and replica 3 is faulty

When the primary p of view $v + 1$ receives $2f$ valid view-change messages for view $v + 1$ from other replicas, it multicasts a NEW-VIEW message to all other replicas

A backup accepts a new-view message for view $v+1$ if it is signed properly, and contains proof that the new primary received enough view change requests.

Thereafter, all replicas are brought up-to-date if necessary in terms of the requests that have been executed and normal functioning resumes.

Recovery. The recovery protocol makes faulty replicas behave correctly again to allow the system to tolerate more than f faults over its lifetime. To achieve this, the protocol ensures that after a replica recovers it is running correct code; it cannot be impersonated by an attacker; and it has correct, up-to-date state.

Recovery is *proactive*: replicas recover periodically independently of any failure detection strategy. A Byzantine-faulty replica may appear to behave properly even when broken; therefore recovery must be proactive to prevent an attacker from compromising the service by corrupting $1/3$ of the replicas without being detected. Our algorithm recovers replicas periodically independent of any failure detection mechanism. However a recovering replica may not be faulty and recovery must not cause it to become faulty, since otherwise the number of faulty replicas could exceed the bound required to provide safety. In fact, we need to allow the replica to continue participating in the request processing protocol while it is recovering, since this is sometimes required for it to complete the recovery.

We assume each replica has a secure cryptographic co-processor, e.g., a Dallas Semiconductors iButton, or the security chip in the motherboard of the IBM PC 300PL. The co-processor stores the replica's private key, and can sign and decrypt messages without exposing this key. It also contains a true random number generator, e.g., based on thermal noise, and a counter that never goes backwards. This enables it to append random numbers or the counter to messages it signs.

Each replica has a *watchdog timer* that periodically interrupts processing and hands control to a *recovery monitor*, which is stored in the read-only memory. For this mechanism to be effective, an attacker should be unable to change the rate of watchdog interrupts without physical access to the machine. Some motherboards and extension cards offer the watchdog timer functionality but allow the timer to be reset without physical access to the machine. However, this is easy to fix by preventing write access to control registers unless some jumper switch is closed.

These assumptions are likely to hold when the attacker does not have physical access to the replicas, which we expect to be the common case. When they fail we can fall back on system administrators to perform recovery.

The first stage of the recovery protocol is to reboot the machine. This is done periodically when the watchdog timer goes off. The recovery monitor saves the replica's state (the log and the

service state) to disk. Then it reboots the system with correct code and restarts the replica from the saved state. The correctness of the operating system and service code is ensured by storing them in a read-only medium. Rebooting restores the operating system data structures and removes any Trojan horses.

After this point, the replica’s code is correct and it did not lose its state. The replica must retain its state and use it to process requests even while it is recovering. This is vital to ensure both safety and liveness in the common case when the recovering replica is not faulty; otherwise, recovery could cause the $f + 1$ st fault. But if the recovering replica was faulty, the state may be corrupt and the attacker may forge messages because it knows the cryptographic keys used to authenticate both incoming and outgoing messages.

The second part of recovery protocol solves these problems by discarding the keys the recovering replica shares with clients and other replicas, and establishing new keys.

The final part of the recovery protocol consists of checking and fetching the service state. While replica i is recovering, it uses an efficient state transfer mechanism based on Merkle trees to determine what pages of the state are corrupt and to fetch pages that are out-of-date or corrupt; details are in [5].

After the recovering replica is brought up-to-date, it can restart the normal execution of service requests.

2.3.2 Implementation

Our algorithm has been implemented as a generic program library with a simple interface. This library can be used to provide Byzantine-fault-tolerant versions of different services.

To evaluate the performance of our algorithm, we ran a number of experiments; the details can be found in [5]. The experiments compare the performance of a replicated NFS implemented using the library with an unreplicated NFS. The results show that the performance of the replicated system without recovery is close to the performance of the unreplicated system. They also show that it is possible to recover replicas frequently to achieve a small window of vulnerability in the normal case (2 to 10 minutes) with little impact on service latency.

2.4 BASE: Using Abstraction to Improve Fault Tolerance

BASE is a replication technique that combines Byzantine-fault tolerance with work on data abstraction [19]. Byzantine-fault tolerance allows a replicated service to tolerate arbitrary behavior from faulty replicas, e.g., behavior caused by a software bug or an attack. Abstraction hides implementation details to enable the reuse of off-the-shelf implementations of important services (e.g., file systems, databases, or HTTP daemons) and to improve the ability to mask software errors.

We extended the BFT library [4, 5] to implement BASE. (BASE is an acronym for BFT with Abstract Specification Encapsulation.) The original BFT library provides Byzantine-fault tolerance with good performance and strong correctness guarantees if no more than $1/3$ of the replicas fail within a small window of vulnerability. However, it requires all replicas to run the same service implementation and to update their state in a deterministic way. Therefore, it cannot tolerate deterministic software errors that cause all replicas to fail concurrently and it complicates reuse of existing service implementations because it requires extensive modifications to ensure identical values for the state of each replica.

The BASE library and methodology correct these problems — they enable replicas to run different or non-deterministic implementations. The methodology is based on the concepts of *abstract specification* and *abstraction function* from work on data abstraction [19]. We start by defining a

common *abstract specification* for the service, which specifies an *abstract state* and describes how each operation manipulates the state. Then we implement a *conformance wrapper* for each distinct implementation to make it behave according to the common specification. The last step is to implement an *abstraction function* (and one of its inverses) to map from the concrete state of each implementation to the common abstract state (and vice versa).

The methodology offers several important advantages.

Reuse of existing code. BASE implements a form of state machine replication [15, 31], which allows replication of services that perform arbitrary computations. But state machine replication requires determinism: all replicas must produce the same sequence of results when they process the same sequence of operations. Most off-the-shelf implementations of services fail to satisfy this condition. For example, many implementations produce timestamps by reading local clocks, which can cause the states of replicas to diverge. The conformance wrapper and the abstract state conversions enable the reuse of existing implementations without modifications. Furthermore, these implementations can be non-deterministic, which reduces the probability of common mode failures.

Software Rejuvenation through proactive recovery. It has been observed [13] that there is a correlation between the length of time software runs and the probability that it fails. BASE combines proactive recovery [5] with abstraction to counter this problem. Replicas are recovered periodically even if there is no reason to suspect they are faulty. Recoveries are staggered so that the service remains available during rejuvenation to enable frequent recoveries. When a replica is recovered, it is rebooted and restarted from a clean state. Then it is brought up to date using a correct copy of the abstract state that is obtained from the group of replicas. Abstraction may improve availability by hiding corrupt concrete states, and it enables proactive recovery when replicas do not run the same code or run code that is non-deterministic.

Opportunistic N-version programming. Replication is not useful when there is a strong positive correlation between the failure probabilities of the different replicas, e.g., deterministic software bugs cause all replicas to fail at the same time when they run the same code. N-version programming [6] exploits design diversity to reduce the probability of correlated failures, but it has several problems [11]: it increases development and maintenance costs by a factor of N or more, adds unacceptable time delays to the implementation, and does not provide a mechanism to repair faulty replicas.

BASE enables an opportunistic form of N-version programming by allowing us to take advantage of distinct, off-the-shelf implementations of common services. This approach overcomes the defects mentioned above: it eliminates the high development and maintenance costs of N-version programming, and also the long time-to-market. Additionally, we can repair faulty replicas by transferring an encoding of the common abstract state from correct replicas.

Opportunistic N-version programming is a viable option for many common services, e.g., relational databases, HTTP daemons, file systems, and operating systems. In all these cases, competition has led to four or more distinct implementations that were developed and are maintained separately but have similar (although not identical) functionality. Since each off-the-shelf implementation is sold to a large number of customers, the vendors can amortize the cost of producing a high quality implementation.

Furthermore, the existence of standard protocols that provide identical interfaces to different implementations, e.g., ODBC [9] and NFS [27], simplifies our technique and keeps the cost of

writing the conformance wrappers and state conversion functions low. We can also leverage the effort toward standardizing data representations using XML.

2.4.1 Implementation

We implemented BASE as a generic program library with a simple interface. In addition, we implemented two prototype services using this technique and studied their performance to provide insight into the overhead introduced by BASE.

The first one is BASEFS — a replicated NFS file system where each replica can run a different off-the-shelf file system implementation. The conformance wrapper and the state conversion functions in our prototype are simple, which suggests that they are unlikely to introduce new bugs and that the monetary cost of using our technique would be low.

We ran the Andrew benchmark to compare the performance of our replicated file system and the off-the-shelf implementations that it reuses. Our performance results indicate that the overhead introduced by our technique is low; BASEFS performs within 32% of the standard NFS implementations that it reuses.

We also used the methodology to build a Byzantine-fault-tolerant version of the Thor object-oriented database [18] and made similar observations. In this case, the methodology enabled reuse of the existing database code, which is non-deterministic.

3 Technology Transfer

This section summarizes the technology transfer through software distributions, personnel, and industrial relationships.

3.1 Software

- JIF. We have implemented a compiler that does a source to source translation from JIF to Java. The JIF compiler is based on our earlier work on PolyJ [20], an extension to Java that supports parametric polymorphism. The JIF compiler makes use of our new techniques for label checking and label inference. This code is available over the web at <http://www.cs.cornell.edu/jif/>
- BFT. We have implemented BFT as a generic program library with a simple interface. The library can be used to provide Byzantine-fault-tolerant versions of different services. The software is available over the web at <http://pmg.lcs.mit.edu/bft/>
- BASE. We have also implemented BASE as a generic program library with a simple interface. The software is available over the web at <http://pmg.lcs.mit.edu/bft/>

3.2 Personnel

Much of the technology has been transferred through students who have worked on the project. Sarah Ahmed, Nick Mathewson, and Zheng Yang all received their MS degrees and left to work in industry. In addition Sameer Ajmani spent nine months working in industry after completing his MS; he has since returned to MIT to work on his PhD.

Andrew Myers and Miguel Castro completed their PhDs doing research under the grant. Andrew worked on the information flow model and is now on the faculty at Cornell. Miguel developed the

BFT protocol; he is now working at Microsoft Research in Cambridge England. Rodrigo Rodrigues received his MS degree for his work on BASE; he is currently working on his PhD at MIT.

4 Publications

The publications for each project are listed below in reverse chronological order.

4.1 Information Flow Publications

- Sameer Ajmani, Robert Morris, Barbara Liskov: A Trusted Third-Party Computation Service. MIT Laboratory for Computer Science Technical Report 847 (2001)
- Sameer Ajmani: A Trusted Execution Platform for Multiparty Computation. Masters thesis, Massachusetts Institute of Technology (2000). Also available as MIT Laboratory for Computer Science Technical Report 846.
- Nick Mathewson: Verifying Information Flow Control in Java. Masters thesis, Massachusetts Institute of Technology (2000)
- Andrew C. Myers, Barbara Liskov: Protecting privacy using the decentralized label model. TOSEM 9(4): 410-442 (2000)
- Andrew C. Myers, Barbara Liskov: Protecting Privacy in a Decentralized Environment. DISCEX 2000: 266-280
- Andrew C. Myers: Mostly-Static Decentralized Information Flow Control. Ph.D. thesis, Massachusetts Institute of Technology (1999). Also available as MIT Laboratory for Computer Science Technical Report 783.
- Andrew C. Myers: JFlow: Practical Mostly-Static Information Flow Control. POPL 1999: 228-241
- Andrew C. Myers, Barbara Liskov: Complete, Safe Information Flow with Decentralized Labels. IEEE Symposium on Security and Privacy 1998: 185-197

4.2 Byzantine-Fault-Tolerance Publications

- Miguel Castro, Barbara Liskov: Practical Byzantine Fault Tolerance and Proactive Recovery. TOCS 20(4): 398-461 (2002)
- Rodrigo Rodrigues, Barbara Liskov, Liuba Shrira: The Design of a Robust Peer-to-Peer System. ACM SIGOPS European Workshop 2002.
- Sarah Ahmed: A Scalable Byzantine Fault Tolerant Secure Domain Name System. Masters thesis, Massachusetts Institute of Technology (2001). Also available as MIT Laboratory for Computer Science Technical Report 849.
- Miguel Castro, Barbara Liskov: Byzantine Fault Tolerance Can Be Fast. DSN 2001: 513-518
- Rodrigo Rodrigues, Miguel Castro, Barbara Liskov: BASE: Using Abstraction to Improve Fault Tolerance. SOSP 2001: 15-28

- Rodrigo Rodrigues, Miguel Castro, Barbara Liskov: Using Abstraction to Improve Fault Tolerance. HotOS 2001: 27-32
- Rodrigo Rodrigues: Combining Abstraction with Byzantine Fault Tolerance. Masters thesis, Massachusetts Institute of Technology (2001). Also available as MIT Laboratory for Computer Science Technical Report 850.
- Miguel Castro: Practical Byzantine Fault Tolerance. Ph.D. thesis, Massachusetts Institute of Technology (2000). Also available as MIT Laboratory for Computer Science Technical Report 817.
- Miguel Castro, Barbara Liskov: Proactive Recovery in a Byzantine-Fault-Tolerant System. OSDI 2000: 273-288
- Miguel Castro, Barbara Liskov: A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. MIT Laboratory for Computer Science Technical Memo 597 (1999)
- Miguel Castro, Barbara Liskov: Practical Byzantine Fault Tolerance. OSDI 1999: 173-186
- Zheng Yang: Byzantine-Fault-Tolerant Secure DNS Infrastructure. Masters thesis, Massachusetts Institute of Technology (1999)

5. References

- [1] S. Ajmani. A trusted execution platform for multiparty computation. Master's thesis, Massachusetts Institute of Technology, 2000. Also available as MIT Laboratory for Computer Science Technical Report 846.
- [2] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as NTIS AD-A023 588.
- [3] M. Castro and B. Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.
- [4] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, Feb. 1999.
- [5] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA, Oct. 2000*.
- [6] L. Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Fault Tolerant Computing*, FTCS-8, pages 3–9, 1978.
- [7] D.E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236-243. 1976.
- [8] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
- [9] K. Geiger. *Inside ODBC*. Microsoft Press, 1995.
- [10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Aug. 1996. ISBN 0-201-63451-1.
- [11] J. Gray and D. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, Sept. 1991.
- [12] M. Herlihy and J. Wing. Axioms for Concurrent Objects. In *ACM Symposium on Principles of Programming Languages*, 1987.
- [13] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, modules and applications. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, pages 381–390, Pasadena, CA, June 1995.
- [14] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Hawaii International Conference on System Sciences*, 1998.
- [15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [16] L. Lamport. The Part-Time Parliament. Technical Report 49, DEC Systems Research Center, 1989.
- [17] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. In *Proceeding of the 13th ACM Symposium on Operating System Principles*, pages 165–182, Oct. 1991. *Operating System Review*, 253(5).
- [18] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In

Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99), Lisbon, Portugal, June 1999.

- [19] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.
- [20] B. Liskov, N. Mathewson, and A. C. Myers. PolyJ: Parameterized types for Java. Software release. Located at <http://www.pmg.lcs.mit.edu/polyj>, July 1998.
- [21] A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228-241 San Antonio, TX, Jan. 1999.
- [22] A. C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, Jan. 1999. Also available as MIT Laboratory for Computer Science Technical Report 783.
- [23] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceeding of the 16th ACM Symposium on Operating System Principles*, pages 129–142, Saint-Malo, France, 1997.
- [24] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*, pages 185-197 Oakland, CA, May 1998.
- [25] A. C. Myers and B. Liskov. Protecting privacy in a decentralized environment. In *DISCEX*, pages 266–280, 2000.
- [26] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *TOSEM*, 9(4):410–442, 2000.
- [27] Network working group request for comments: 1094. NFS: Network file system protocol specification, March 1989.
- [28] B. Oki and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing*, 1988.
- [29] M. Reiter. The Rampart Toolkit for Building High-Integrity Services. *Theory and Practice in Distributed Systems (LNCS 938)*, 1995.
- [30] R. S. Sandhu. Role hierarchies and constraints for lattice-based access controls. In *Proc. Fourth European Symposium on Research in Computer Security*, Rome, Italy, Sept. 25-27 1996.
- [31] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.